Don't trust your USB! How to find bugs in USB device drivers

Sergej Schumilo
OpenSource Security Ralf Spenneberg
University of Applied Sciences Muenster
schumilo@fh-muenster.de

Ralf Spenneberg
OpenSource Training Ralf Spenneberg
ralf@spenneberg.net

Hendrik Schwartke
OpenSource Security Ralf Spenneberg
hendrik@spenneberg.net

Abstract

Attacks via USB have been demonstrated regularly for some years. A comprehensive systematic analysis of the potential risk of this weakness was very expensive, yet. The framework presented in this paper supports the systematic high performance fuzzing of USB drivers using massive parallel virtual machines. The framework identifies and documents the security weaknesses. The format used permits the reproduction of the attack, supporting the analysis by developers and analysts.

Keywords: USB, Fuzzing, QEMU, KVM, USB-Redirection, Emulation

1. Introduction

The Universal Serial Bus (USB) has been an attractive security hot spot for some years. Well known attack vectors have been the USB autorun functionality of older Windows versions or the creation of malicious HID-devices using the Teensy framework [1], which uses shortcuts and keyboard emulation to compromise systems. At least since the publication of the BadUSB[2] tool, the conceptional security weaknesses embedded in USB have been known.

Further attacks are possible by exploiting the implementation errors in USB drivers. Every USB device needs either a generic driver based on the device class (e.g. HID devices like keyboards) or device specific drivers. These drivers are often vendor-specific drivers. The driver loaded by the operating system is responsible for the communication with the device. This driver is usually loaded within kernel space and often trusts the device. This trust can be exploited by using specific USB fuzzing solutions like the popular Facedancer board [3].

The USB fuzzing technique either tries to specifically modify the USB traffic implementing a Man-in-the-Middle approach[4] or to emulate a malicious USB device, which injects malicious payloads into the USB host. In the past, no approach supported a systematic, comprehensive analysis of all available USB device drivers of a given operating system. The vast amount of possible USB device drivers and the possible variations of the USB traffic require the generation of several million possible fuzzing tests. The slow speed of the known hardware solutions (Facedancer, etc.) makes such fuzzing attacks virtually impossible.

This papers presents the development of USB fuzzing framework named vUSBf (virtual USB fuzzer). This framework increases the speed of the fuzzing test by several magnitudes. Additional features support the reproducibility and analysis of the identified weaknesses.

2. Solution

The main challenges in the implementation of the fuzzing framework are the speed, high reproducibility and complete logging of the fuzzing tests, using different USB payloads. A further requirement is the simple extensibility of the framework with additional features. The following technologies permitted the design and implementation of a framework fulfilling these requirements and challenges.

2.1. Host virtualization

The framework builds upon the Linux kernel integrated hypervisor KVM. Using this hypervisor, the framework virtualizes the tested operating systems. The USB infrastructure is emulated using QEMU. The open source license permits the simple and flexible extension by our own patches and addons.

2.2. USB virtualization

We deploy the USB redirection protocol for the virtualization of the USB protocol. The USB redirection protocol is used to provide access to remote USB devices in virtual desktop infrastructures. Currently only QEMU supports the USB redirection protocol [5]. This interface enables the transport of USB traffic via TCP, UDP or Unix sockets.

Part of the USB redirection suite is the usbredirserver. This software makes a USB device available on the network. On the host system, the physical USB device is bound by its own driver via libusb. All requests by the QEMU guest are transferred via the USB redirection protocol to this driver, resulting in a logical connection between the QEMU guest and the USB device.

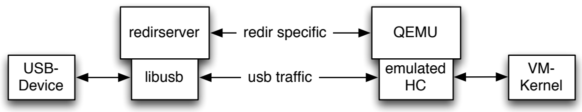

Figure 1. Usage of the usbredirserver

Logically the USB traffic is transferred between libusb and the emulated host controller (HC) on the guest (Fig. 1). The USB redirection protocol encapsulates the messages. This encapsulation is accomplished by the usbredirserver and the appropriate interface in QEMU.

The simplicity of the USB redirection protocol permits the eavesdropping and manipulation of the connection at low cost. Thus, USB fuzzing without deep knowledge of the USB protocol is possible. Additionally, different USB devices may be emulated using the USB redirection protocol. This makes stateful USB fuzzing feasible without using physically connected hardware USB devices. Scalability is improved considerably.

A further advantage of the USB redirection protocol is the missing limitation of the number of the available endpoints. Emulated devices may possibly support all 31 endpoints supported by USB. The USB redirection protocol is unaware of the used USB protocol. Future emulations may even support USB 3.0.

2.3. Harddrive Virtualization

Deploying virtual machines using QEMU implies the usage of image files or real filesystems to provide the virtual harddisks for the guests. We are using QCOW2 images for flexibility and speed. The QCOW2 format supports copy on write and overlay files. A simple guest setup is illustrated in fig. 2.

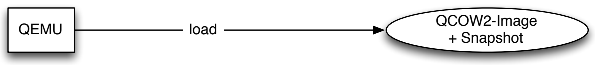

Figure 2. Simple setup using QEMU

The QCOW2-Image supports the storage of snapshots. These snapshots store the exact state of the guest system including delta of harddisk operations. Instead of booting the guest system, the guest may be initialized instantaneously using a known stored state. The QCOW2 image is at least enlarged by the amount of the guest's main memory.

Using many guests, the size of the snapshot files becomes very important. Our fuzzing framework uses many parallel QEMU guest instances, each requiring its own backing files. This would demand a huge amount of file space and would hinder effective file caching.

To bypass this disadvantage, we are proposing the utilization of overlay files. Overlay files are connected to a designated backing file and store just the delta of harddisk operations. The original backing file is untouched and just used for read operation. All write operations are directed at the overlay file (copy on write). Using overlays, every QEMU guest instance would access the same large backing file and all modifications would be written to the initially small overlay file (fig. 3).

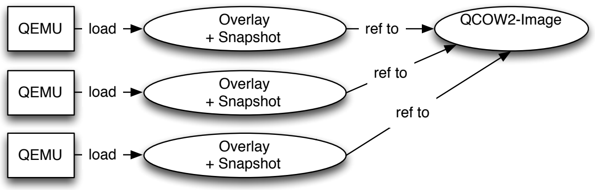

Figure 3. Using overlays

Unfortunately when using overlays, snapshots created in the original backing file are not accessible anymore. Thus each overlay would need its own snapshot, increasing the overlay file again by the amount of main memory. This would again result in larger overlay file sizes.

Tests proved that QEMU supports the usage of several virtual harddisks. The system state snapshot is always stored in the image file defined first on the command line. Copy on write operations are handled separate for each harddisk. Of course the snapshot may only be reinitialized if all virtual images are available.

Our fuzzing framework therefore uses the concept shown in fig. 4.

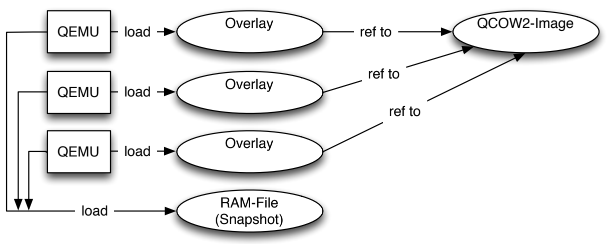

Figure 4. Overlays supporting best memory usage

When cloning virtual machines for fuzzing tests this concept requires only one small unique image file per virtual machine. Both the harddisk backing file and the file containing the snapshot are only read and can be reused for all virtual machines. The only written file is the overlay containing the copy on write operations during the fuzzing test. All files will probably be aggressively cached by the host system. Alternatively these files may be moved to a ramdisk because of their small footprint.

2.4 Reproducibility

We developed our own fuzzing engine. This engine permits the simple reproduction of the USB fuzzing tests. Although the used Scapy framework[6] implements its own fuzzing methodology, we refrained from using Scapy for the fuzzing. Scapy uses random values in its fuzzing, preventing the fine-tuned control and documentation of the single fuzzing tests.

Our fuzzing engine uses testcases. Each fuzz testcase is referenced by a unique ID and contains the fuzzing instructions. Lists of testcases may be combined using mathematical conjunctions like the union sets (join of two lists) or the pairwise union (for each combination of all testcases). The goal is the simple automatic generation of a large amount of testcases supporting systematic and comprehensive fuzzing. A testcase may be anything from a single fuzzing instruction to a combination of an arbitrary number of other testcases.

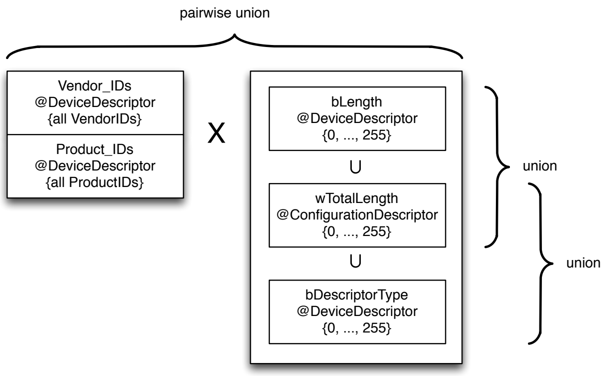
pairwise union
Vendor_IDs @DeviceDescriptor {all VendorIDs}
Product_IDs @DeviceDescriptor {all ProductIDs}
X
bLength @DeviceDescriptor {0, ..., 255}
U
wTotalLength @ConfigurationDescriptor {0, ..., 255}
U
bDescriptorType @DeviceDescriptor {0, ..., 255}
union
union

Figure 5. Fuzz unit combination

A simple example (fig. 5) illustrates the advantages of this approach. A set of all possible vendor and product ids in the device descriptor is combined with a second set of all possible fuzzing instructions important in the enumeration phase of USB. All device drivers referenced by the vendor and product IDs are thus systematically tested. The actual description of the fuzz units and their application is defined in XML files (fig. 6).

The simple example may be extended by further fuzz units at one's discretion. The conjunction may optionally be adjusted to combine only those vendor and product ids used by mass-storage devices with SCSI specific fuzz units. A list of all well known vendor and product ID values and further class specific ids may be found in the USB database on linux-usb.org [7].

Each testcase or a sequence of testcases may be exported to a file. This file may be loaded and the stored testcases may be replicated to trigger the found bugs again. This permits the demonstration of the found weaknesses and the testing of their successful fix.

```
...
    <testcases name="testcases_1" >
        <testunit type="pairwise_union" >
            <fuzz_list name="all_vendor_product_ids" />
            <testunit type="union" >
                <fuzz_list name="dev_desc_blength_invalid" />
                <fuzz_list name="conf_wTotalLength_invalid" />
                <fuzz_list name="dev_bDescriptorType_invalid" />
            </testunit >
        </testunit >
    </testcases >
...
```

Figure 6. Contents of testcase.xml

3. Implementation

The vUSBf framework is implemented in Python. The architecture of the framework is shown in the following illustration (fig. 7).

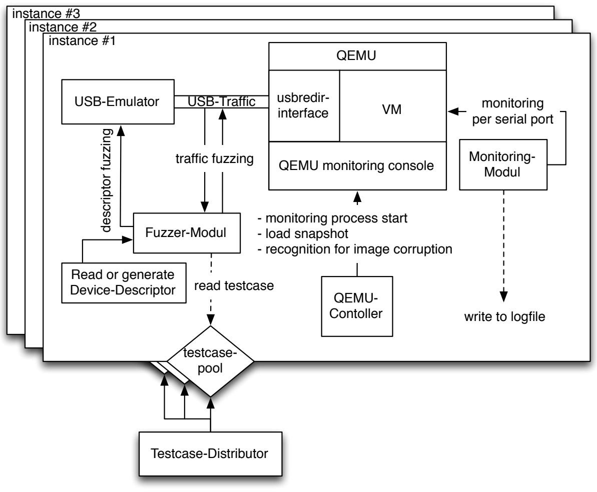

Figure 7. Architecture of the framework

3.1 QEMU Controller

The main focus of the framework is the supervision and monitoring of the running QEMU processes.

Thus a very important item of the framework is the QEMU controller. The QEMU controller runs the individual QEMU instances as subprocesses. The monitoring of these processes happens via standard input and output of the QEMU process. The QEMU-controller monitors the output of the QEMU process for any possible error messages. A typical known error message reports on the possible corruption of the image file. The QEMU built-in monitoring console supports the initialization of snapshots or the termination of the QEMU process. As soon as the QEMU controller notices an error in the execution of the virtual machine, the Controller can correct the error and restart the process.

3.2 USB Emulation & Fuzzing

The USB emulation is implemented using an extendable API. The USB communication is handled by a unique Unix socket per virtual instance. First the emulator for the USB redirection protocol is started. This emulator passes the command on to the emulator for the chosen testcases. Every emulator inherits the capabilities of the enumeration emulation class and extends this class with its own device-specific handling.

The USB fuzzing is implemented using the Scapy framework. Each emulator operates on a Scapy object representing the USB message. This object is passed on to the API function send(). This function works as a wrapper and relays the Scapy object to the fuzzing engine. The fuzzing engine obtains a testcase from the defined pool of testcases and applies all the contained fuzzing instructions to the USB message. The modified message is returned to the wrapper which subsequently dispatches the outgoing message. Incoming USB traffic is not manipulated.

Additionally the emulator supports the fuzzing of device descriptors. The structures embedded in the device descriptors may be modified. For example, a device descriptor may be extended by an additional endpoint descriptor or specific areas of a given device descriptor may be deleted.

3.3 Monitor

Fuzzing is only successful if the victim system is under surveillance and the impact can be correlated with the data used in the fuzzing test. We use an interchangeable monitoring module for the observation of the victim system. Depending on the examined operating system, different approaches are required. Linux systems may easily be monitored via a virtual serial port connected to a TTY of the system. If the monitored Linux system is configured to report kernel messages via printk on this console all kernel messages may be read. By setting the verbosity to 7 (highest) via **echo** '7' > /proc/sys/kernel/printk even stack traces and verbose error messages are reported. The Linux monitoring module collects and reports this information in combination with the applied testcase ID in log files. Additionally the monitoring module is notified by the USB emulator as soon as the USB stack becomes unresponsive. This is again logged including the testcase ID. Furthermore the QEMU controller is notified to reload the snapshot or restart the QEMU process. Depending on the configuration, additional objects are written on disk which may be reloaded by the framework to reproduce and test the identified bug.

The monitoring module for Microsoft Windows is still in development. Unfortunately, Windows does not support the simple output of kernel messages on serial consoles. Currently different approaches like the analysis of the average color of Windows screenshots to distinguish bluescreens or using WinDbg are evaluated.

3.4 Multiprocessing & Clustering

A major feature of the framework is the parallel execution of virtual machines using multiprocessing. Using several parallel virtual machines, the number of the tests can be enhanced. The framework uses a testcase distributor to supervise the autonomous QEMU controllers. The testcase distributor uses IPC to communicate with the different QEMU controllers on the same host[1].

[1]First implementations using multithreading have been found impossible in Python because of the Global Interpreter Lock[8]

To further enhance the scalability, the vUSBf Framework offers a network interface for clustering. The vUSBf framework implements a simple TCP based protocol. The master testcase distributor connects to all testcase distributors, feeds them testcases and coordinates the tasks among several testcase distributors. The vUSBf framework simply scales across several physical hosts and is only limited by the amount of resources available.

4. Performance

The framework supports USB fuzzing in two modes: “reload” and “reload if needed”. The first mode reloads the virtual machine by loading the snapshot after each unique fuzzing test. The test always starts in exactly the same state. The second mode is comparable to the fuzzing using a hardware solution like the Facedancer. Here the system is attacked using malicious USB payloads until the system becomes unresponsive or crashed. This may find additional bugs only triggered by the combination of such tests.

Naturally, the effective fuzzing performance depends on the mode. The following benchmarks were executed on a single server using 4 Intel Xeon E5-2630L CPUs with 6 cores each and 64GB RAM.

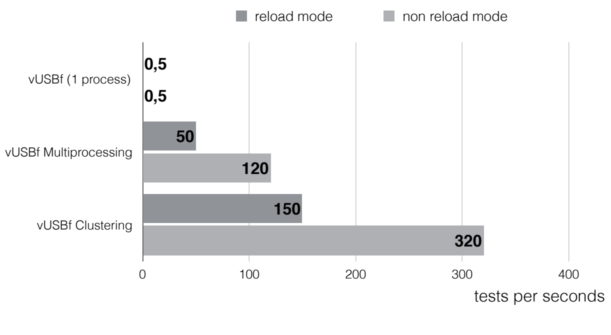

Figure 8. Tests per second performance

Depending on the number of testcases, the performance may be greatly enhanced. The following images illustrates this for 1,000,000 testcases.

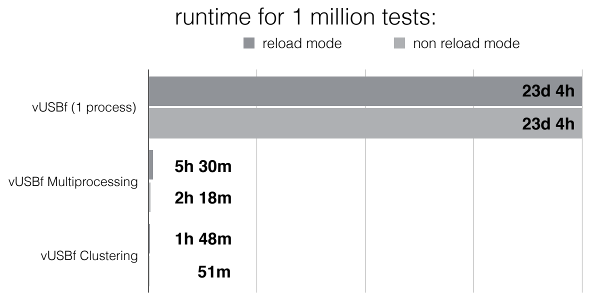

Figure 9. Runtime performance

5. Results

So far, many different bugs have been found in current Linux kernel versions. Most of these bugs were found in a small portion of the device drivers.

The current version of the framework implements the emulation of the USB enumeration phase and HID devices. Therefore all currently identified bugs are located in this area. During the enumeration phase the transferred information is parsed by the Linux USB core driver and depending on the vendor and product ID or the USB class ID the information is passed on the specific driver. Therefore, all bugs refer to the initialization of these drivers. We trust that further bugs will be identified as soon as additional emulators are available.

Currently, the following bugs have already been identified by only fuzzing the device descriptors:

- Null-pointer dereferences
- Kernel paging requests
- Bad page state
- Segfault
- Kernel panic

The mode ”only reload if needed” even unearthed bugs only triggered by a specific combination of loaded drivers. Vulnerable drivers are udlfb, r8192u_usb and hfa384x, which crash the system and generate kernel panics. Although some of the identified drivers currently are in staging state most Linux distributions enable and ship these drivers. Testing different kernel versions, we also detected new introduced bugs not present in older versions of the Linux kernel.

To verify these findings, some randomly chosen bugs have been implemented using the Facedancer. Using the Facedancer, we were able to prove that the bugs identified by our framework may be reproduced on physical systems and are not due to the virtualization framework used in our tests.

Because of time constraints most bugs have not yet been analyzed. An assessment of their severity is currently not possible. To support the analysis, the framework can be utilized to reproduce any payload or even a sequence of payloads. The virtual system may then be analyzed using typical tools like the KGDB interface to connect to the crashed kernel and debug the system.

```
Call Trace:
 [<ffffffff815271fa>] ? panic+0xa7/0x16f
 [<ffffffff8152b534>] ? oops_end+0xe4/0x100
 ...
```

Many bugs found by the framework generate “only” Linux kernel Oops. These Oops only affect kernel subsystems and not the complete Linux kernel. Thus, only a denial of service of the subsystem may be triggered by the bug. Enterprise Linux distributions (RHEL, CentOS) often use a build option “Panic-on-Oops” which triggers a kernel panic on each Oops. These systems are then vulnerable to a full denial of service (DoS).

6. Future Work

Currently the framework implements a proof of concept. We are still actively developing the framework and adding further features. All interested researchers are invited to test the framework, analyze the identified bugs and suggest further features or enhance the framework. The current version of the framework may be retrieved via git[2].

Planned features are the support of Microsoft Windows and additional specific emulators. These emulators will support driver specific emulation and fuzzing. The planned emulators will support the USB classes Mass−Storage and Printer . Fuzzing USB 3.0 device drivers is planned as well.

7. Conclusion

The presented fuzzing framework vUSBf permits much higher fuzzing execution speeds than all comparable solutions in the past. Any operating system supported by KVM and QEMU may be analyzed. The analysis performed by the framework is systematic and comprehensive. Systems not supported by the framework still need to be analyzed using hardware solutions like the Facedancer.

We are convinced that using the systematic analysis via vUSBf bugs and weaknesses will be found in various systems. Several of these bugs may be exploited for denial of service or even compromise of the system using malicious USB devices. Using the export and reproduction feature of the framework, both security researcher and device driver developer may work hand in hand in the identification and patching of the found weaknesses.

References

[1] A. Crenshaw, “Programmable HID USB Keystroke Dongle.” https://www.defcon.org/images/defcon-18/dc-18-presentations/Crenshaw/DEFCON-18-Crenshaw-PHID-USB-Device.pdf, 2010. [Online; accessed 10-September-2014].

[2] S. K. Karsten Nohl and J. Lell, “BadUSB — On accessories that turn evil.” https://srlabs.de/blog/wp-content/uploads/2014/07/SRLabs-BadUSB-BlackHat-v1.pdf, 2014. [Online; accessed 28-September-2014].

[3] T. Goodspeed and S. Bratus, “Emulating USB Devices with Python.” http://travisgoodspeed.blogspot.de/2012/07/emulating-usb-devices-with-python.html, 2012. [Online; accessed 10-September-2014].

[4] R. van Tonder and H. Engelbrecht, “Lowering the USB Fuzzing Barrier by Transparent Two-Way Emulation, USENIX WOOT’14,” 2014.

[5] H. de Goede, “USB-Redirection protocol information.” https://raw.githubusercontent.com/SPICE/usbredir/master/usb-redirection-protocol.txt, 2014. [Online; accessed 10-September-2014].

[6] P. Biondi, “Scapy.” http://www.secdev.org/projects/scapy/.

[7] “The USB ID Repository.” http://www.linux-usb.org/usb-ids.html. [Online; accessed 10-September-2014].

[8] “Global Interpreter Lock at wiki.python.org.” https://wiki.python.org/moin/GlobalInterpreterLock. [Online; accessed 10-September-2014].

[2]https://github.com/schumilo